A Framework for Program Development Based on Schematic Proof

A Framework for Program Development Based on Schematic Proof

David Basin[1], Alan Bundy[2], Ina Kraan[2], and Sean Matthews[1]

[1] Max-Planck-Institut für Informatik
Im Stadtwald
D-66123 Saarbrücken, Germany
{basin,sean}@mpi-sb.mpg.de
[2] Department of Artificial Intelligence
University of Edinburgh
Edinburgh, Scotland, U.K.
{bundy,inak}@ai.ed.ac.uk

Abstract. Often, calculi for manipulating and reasoning about programs can be recast as calculi for synthesizing programs. The difference involves often only a slight shift of perspective: admitting metavariables into proofs. We propose that such calculi should be implemented in logical frameworks that support this kind of proof construction and that such an implementation can unify program verification and synthesis. Our proposal is illustrated with a worked example developed in Paulson's Isabelle system. We also give examples of existent calculi that are closely related to the methodology we are proposing and others that can be profitably recast using our approach.

1 Introduction

What is the difference between program verification and program synthesis? Can a calculus designed for one of these activities be reused or recast for the other? These two questions motivate our work here; their answers are important as they help us not only to understand better the theoretical relationship between these development paradigms, but also to develop calculi and implement them on machines.

Let us illustrate the relationship between these two approaches, and that there exists a space of design options between them, by choosing a popular kind of programming logic as an example: constructive type theory as a logic for functional programs, as embodied in Martin-Löf's type theory [26], Nuprl [10] or the Calculus of Constructions [11]. In such logics one proves that a term t meets a specification T by demonstrating that t has the type T, denoted $t \in T$. For example, if

$$t \in (\forall x{:}T_1.\, \exists y{:}T_2.\, R[x,y])$$

then, by the nature of the logic, for any x in in T_1, $t(x)$ evaluates to a pair $\langle f(x), p(x)\rangle$ in T_2 where $p(x)$ is a proof that $R[x, f(x)]$. If $R[x,y]$ were the relation that y is a sorted version of x, then $f(x)$ must be a sorting program.

Whether such a type theory is used for verification, synthesis, or both, depends on how its rules are formalized. Martin-Löf, in presenting his logic, gives rules for demonstrating that terms in the theory are members of types; that is, the rules construct verification proofs (showing a given t meets some specification T). The Nuprl system provides similar rules as "refinement rules": one starts with a goal T and constructs a proof that T is true. In such proofs, the program t may or may not be given a priori, and the system contains two (very similar) sets of rules depending on whether t is present (i.e., the goal is $t \in T$) or not (i.e., the goal is simply T). In the former case, proofs verify programs. In the latter, they say how to construct or synthesize programs, and the system may *extract* the inhabiting term t. The synthesis and extraction may be understood in a very simple way: the term t is initially a metavariable, and each proof rule elaborates a bit more of the structure of t (using substitutions derived from unification). The Oyster system [6] uses exactly this approach and manipulates these metavariables behind the scenes. In [28], Paulson sketches an implementation in Isabelle of Martin-Löf's calculus. Here the relationship between verification and synthesis is especially clear: they are one and the same activity! That is, metavariables become "first-class" objects and may appear directly in proofs as well as proof rules; hence one reasons about a term $t \in T$ where t may be a specific ground term of the type theory, a metavariable, or even a combination of both. In the first case we have verification, in the second synthesis, and in the third a hybrid where some parts of the program structure are known and others left unspecified (see [17] for examples of uses for this). In all three cases the same kind of proof rule is used to build proofs, independent of whether the goal contains an actual or a schematic term; hence verification and synthesis are unified.

This idea of unifying verification and synthesis through metavariables is not new. It goes back at least as far as Green's use of resolution not only for checking answers to queries, but also for synthesizing programs [13]. In his work, metavariables were introduced through Skolemization in preparation for resolution (see section 4 for further details).

In this report we suggest that the second-order generalization of this idea can be applied to unify program verification and synthesis in a wide range of settings. Our contributions will be several fold: first, we give this generalization, justify why it is appropriate, and explain how it can be simply implemented using the Isabelle theorem prover (section 2); second we present a worked example of the kind of development paradigm we have in mind, that of logic program synthesis (section 3); finally, we conclude with a brief survey of related work and give examples of existent calculi that are closely related to the methodology we are proposing and show how others can be profitably recast using our approach.

2 Schematic Proofs

2.1 Modeling Proof Rules and Deductions

We are proposing an approach for implementing program synthesis calculi, so it is fitting that we begin by considering some of the design issues involved which lead us to our choice. This will also help us compare our work with other approaches later.

In logic texts, proof rules, and sometimes axioms, are presented schematically. That is, they contain metavariables ranging over terms and formulae. For example, in a proof rule like

$$\frac{A \quad B}{A \wedge B} \qquad (\wedge\text{-I})$$

A and B are not formulae but variables ranging over formulae. On the other hand, aside, perhaps, from when one is doing metatheory, one works only with ground terms.[1] For our work we must allow both proof rules and proofs themselves to be schematic. What is involved in providing machine support for this?

For logics without operators that bind variables (e.g., quantifiers) the answer is easy: we can represent both proof rules and proofs using terms that may contain first-order metavariables. That is, the metavariables range over the syntactic categories of the logic (terms, formulae, etc.) and may be manipulated using (sorted, if there is more than one category) first-order unification. So in the above rule for $\wedge$-I, A and B may be first-order metavariables ranging over, say, propositional logic terms. Another example would be Prolog, where a program is a set of axioms with schematic terms and execution corresponds to building a proof that can contain schematic terms.

When logical syntax employs operators that bind variables, first-order metavariables are insufficient. Consider for example the rule

$$\frac{\forall x.\, A}{A[t/x]}. \qquad (\forall\text{-E})$$

If we tried to represent the schematic formula in the premise of this rule as $\forall x.\, A$, where A is a first-order metavariable and substitution is as usually defined in first-order logic, then we could neither adequately instantiate A nor properly substitute t for x. The former is problematic because substitution should be capture avoiding (how is A to reference the name of the bound variable — which in this case is x?); the latter is problematic because first-order substitution is

[1] We employ the following terminology: A logical framework like Isabelle provides a *metalogic* for encoding and reasoning about *object logics*. *Metavariables* are variables in the metalogic which range over the syntactic categories of the object logic. We call terms containing metavariables *schematic terms* and proofs containing metavariables *schematic proofs*. Schematic terms are also used in in informal mathematics, although the metalogic is unspecified. We will exclusively use the term *ground* to refer to terms which contain no metavariables, although they may contain variables of the object logic (either free or bound).

only defined on ground terms. Of the few proof systems that allow metavariables in proofs (e.g., [12,23,29]) higher-order metavariables are used and object-level syntax is encoded using some kind of higher-order abstract syntax. A notable exception is the KIV system[16], which possesses both first-order metavariables and binding operators in the logic; it copes with the above mentioned instantiation problems somewhat crudely: substitution for metavariables allows capture and substitution for ordinary variables is allowed only in ground terms; although this preserves validity of proofs, it greatly restricts the way in which they may be built.

One option for implementing schematic rules and proofs on a machine is to formalize an appropriate notion of variables ranging over terms with holes and their interaction with binding operators. Such a calculus has been formalized in [31]. A simpler approach, used by advocates of "higher order abstract syntax", is to use variables ranging over functions in the lambda calculus. Under this approach a schematic term like $A[x]$ is represented as an application $A(x)$ where A ranges over functions.[2] With this representation, given a term like $\forall x.\, A(x)$ we may instantiate A with a formula valued function, say $\lambda y.y+3=5$, and perform a β-reduction yielding $\forall x.\, x+3=5$. Now applying $\forall$-E with some (perhaps schematic) argument t yields $t+3=5$. This is identical to first instantiating $\forall x.\, A(x)$ with t (yielding $A(t)$) and then instantiating A. The point is that this representation allows us proofs (and proof rules) where each step is valid for all instantiations. Hence, we may construct a program incrementally through a proof and the result is logically the same as if we had first given the program and then verified it (with the same proof!).

Implemented languages and frameworks supporting higher-order metavariables and unification, such as λ-Prolog or ELF, could provide an implementation to base our work upon. We have chosen Isabelle as it is well suited for interactive proof construction, manipulation of schematic rules and proofs, and has support for automated proof construction. These points will become clearer with our example.

2.2 Construction of Synthesis Proofs

We have conducted a number of synthesis experiments in the Isabelle system. These include synthesizing logic programs, functional programs, and representations of circuits. In this section we provide details on how Isabelle can be used this way, and background necessary to understand our worked example. An detailed description of Isabelle can be found in [28].

For the kinds of examples we have in mind, it helps to begin with the view of a *judgement* or assertion. Most proof systems have only one judgement, provability, but one can imagine other kinds; e.g., Martin-Löf has four in his type theory (typehood, equality of types, membership in a type, and equalities of members). We will use this notion to capture the idea that sequents we manipulate have

[2] Functions whose domain is the syntactic category of x and co-domain the syntactic category of $A[x]$.

certain shapes. For example, if we wish to reason about imperative programs using a Hoare logic, then the judgements are triples of the form $\{S\}\,P\,\{Q\}$ where S and Q are first order formulae and P a "while-loop" program. In this case, proof rules will be structured so that they unify with such triples. Another example is that of reasoning about VLSI or gate-level circuits: the judgements there are

$$\forall\, \overline{x}.\, spec(\overline{x})\ R\ Prog(\overline{x})$$

where $\overline{x}$ is a vector of variables representing port values, $Prog$ is a term (or metavariable) representing a circuit with external ports among the $\overline{x}$, *spec* is a specification in first (or higher-order) logic expressing constraints on the $\overline{x}$, and R is a *refinement* relation expressing the relationship between the program and the specification. Typically such relations are equivalence or implication (when the program may be more "concrete" than the specification). We will continue this discussion of hardware synthesis in section 4.

We wish to formalize such judgements in Isabelle and give proof rules such that proofs will verify or construct (depending if there are metavariables in the original conjecture) programs that are correct with respect to the intended semantics of the given programming logic. Isabelle is well suited to support this activity. First, Isabelle is a logical framework. This means we can declare within it a desired object language (syntactic categories and constructors) and axiomatize its proof rules. For instance, in first-order logic we would declare function constants like implication and universal quantification respectively of types $o \to o \to o$ and $(i \to o) \to o$, where i is the sort of first order individuals, and o the sort of formulae. This logic is then axiomatized by giving introduction and elimination rules for these defined logical constants.

Isabelle provides direct support for the construction of schematic proof rules and proofs. It manipulates directly schematic inference rules of the form $[\![\phi_1; \ldots; \phi_n]\!] \Longrightarrow \phi$ where the ϕ may contain higher-order metavariables. The $\Longrightarrow$ is meta-level implication and this sequent represents the formula $\phi_1 \Longrightarrow (\ldots \Longrightarrow (\phi_n \Longrightarrow \phi)\ldots)$ in Isabelle's metalogic, which is a fragment of higher-order logic. (The reader should be careful not to confuse Isabelle's implication, $\Longrightarrow$, with implication in the object logic, or its higher order universal quantifier "!!" with quantifiers in the object logic.) A proof in Isabelle proceeds by applying derived rules to formulae of the above form until all the ϕ_i are proven, at which point ϕ is also proven. In our work, ϕ may initially contain metavariables, and at the end we can read off an assignment for them from the final proof. This is illustrated below.

3 An Example: Logic Program Synthesis

We have chosen this problem domain as it admits a simple exposition and illustrates the essential ideas of our proposed methodology: with only a few simple derived inference rules we may synthesize interesting logic programs from first-order logic specifications. This domain is also interesting in its own right. Even though full first-order logic can be seen as a programming language (e.g., via transformation to normal programs [22]), there are many benefits to having a

logic for deriving logic programs from first-order specifications. For example, we may wish to synthesize an efficient program by changing the underlying algorithm embodied in the recursive structure of the specification. Synthesizing a quick-sort sorting algorithm from a "slow sort" specification, $perm(x,y) \wedge ord(y)$, is an example of this. Alternatively, we may wish to alter the way an algorithm is represented, for example converting a normal program (in the sense of [21], i.e., one which may contain negative atoms in the program body) to a Horn clause program. Such a transformation could prevent floundering during Prolog execution. This approach to logic program development is described in detail in [19,20]; these papers describe the use of the CLAM theorem prover to synthesize programs from their specifications automatically.

We must begin by fixing a programming logic and relationship between programs and specifications. We choose first-order logic as a specification language and say a program is correct when it is equivalent to its specification. Actually, this is not enough: even for logic programs without impure features there are many rival notions of equivalence. The differences though (see [24,5]) are not relevant from the standpoint of illustrating our methodology. The notion of equivalence we choose is to associate a set of Horn clauses with their Clark Completion [21] and show its equivalence to its specification in predicate calculus with theories of standard data-types (e.g., numbers or lists). The details of this follow.

We define a *Horn body* to be a formula as follows

$$G ::= A \mid G_1 \wedge G_2 \mid G_1 \vee G_2 \mid \exists x.\, G$$

where A is a set of atomic predicate names (e.g., known relations like $=$, $\neq$, $TRUE, FALSE$ and previously defined programs including the one currently being defined). We define a *Horn Program* to be a formula of the form $name(\overline{x}) \leftrightarrow body(\overline{x})$ where *body* is a Horn body, *name* is an atomic predicate with variables $\overline{x}$ and the free variables of *body* are contained in $\overline{x}$. This definition guarantees that $name \leftarrow body$ is a Horn clause (in the sense of [27]); such a definition may be straightforwardly translated to a set of standard Prolog Horn clauses. As an example, the following relation *sum* is a Horn program:

$$\begin{array}{l} sum(x,y,z) \leftrightarrow (x = 0 \wedge y = z) \vee \\ \quad \exists x'.(succ(x') = x \wedge ((z = 0 \wedge FALSE) \vee \\ \quad\quad \exists z'.(succ(z') = z \wedge sum(x',y,z')))) \end{array}$$

And it translates directly into the following Prolog program:

```
sum(0,Y,Y).
sum(succ(X),Y,succ(Z)) :- sum(X,Y,Z).
```

(note that the program may be run in various modes; i.e., it can not only add, but also subtract or find pairs of numbers summing to a third).

We will reason about judgements of the form $\forall\, \overline{x}.\, (spec(\overline{x}) \leftrightarrow prog(\overline{x}))$, where *spec* is a concrete first-order specification and *prog* is a schematic Horn program.

As first-order logic is our specification language, verification will take place in Isabelle's standard theory of first-order logic augmented with axioms for natural numbers and lists. These are standard Isabelle theories and come with the system along with tactic support for rewriting. For example, we could prove in this theory, by induction on x, that

$$\forall x\, y\, z.\, (x + y = z \leftrightarrow sum(x, y, z)),$$

where *sum* is the program given above. Alternatively, we could begin with a metavariable for *sum* and synthesize it from such a proof. We show this shortly.

Proof Rules for Program Synthesis

Given a schematic goal, there are two ways we might prove it. The first corresponds to verification: instantiate the goal with a program and proceed from there to show the equivalence. Indeed, given a goal $spec(\overline{x}) \leftrightarrow P(\overline{x})$ the first thought that might occur to us is to complete the proof in one step, by instantiating the metavariable P with *spec*. If our specification language does not satisfy the same restrictions as our programming language (and ours does not) then arbitrary instantiation cannot be admitted; we must insist that P is instantiated with a Horn program. Such constraints are not unique to our work and we will see examples in Section 4 of such side conditions. In our case, we must insist that any instantiation yields a Horn body. Note, furthermore, that even when *spec* is already a Horn body we may still wish to delay instantiation and instead try another proof strategy to synthesize a more efficient program. This is analogous to program transformation work (e.g., the "fold-unfold" technique of [7]) where the initial program specification is already executable but the program is manipulated to derive a more efficient program.

Usually, instead of reducing synthesis to verification, we proceed by decomposing the specification using rules that at the same time preserve the necessary syntactic properties of the synthesized program (i.e., never introduce non-Horn structure during metavariable refinement). Let us give an example of such a rule. Our Isabelle theory contains an inductively defined type for the natural numbers built from 0 and successor `succ` along with Peano axiomatization. We can use this to derive rules that construct Horn programs that recurse or case split depending on their argument values. The following is a simple example of introducing a case split in a Horn program depending on whether a number is `0` or not:

```
Case Split:
[|  ALL y. B(0,y) <-> C(y); !!n. ALL y. B(succ(n),y) <-> D(n,y) |]
==> ALL x y. B(x,y) <-> (x=0 & C(y)) | (EX n.succ(n)=x & D(n,y))
```

A few explanations are in order. `ALL` and `EX` are universal and existential quantification in the encoded logic and, as previously mentioned, should not be confused with "!!" which is universal quantification in Isabelle's metalanguage. Similarly

`&`, `|`, and `<->` are object-level conjunction, disjunction, and equivalence as opposed to meta-level connectives like `==>`. The rule is initially postulated with free variables `B`, `C`, and `D`; this prevents their premature instantiation during proof (which would lead to a proof of something more specialized). When the proof is completed, these variables are replaced by schematic variables (whereby each is prefixed with a "`?`" which is Isabelle's way of denoting metavariables).

Reading the above rule as a refinement rule (i.e., a way to refine a goal into subgoals that imply the original goal) it says that to prove that `B(x,y)` is equivalent to `(x=0 & C(y)) | (EX n.succ(n)=x & D(n,y))` it suffices to prove two cases: one where `B(0,y)` is equivalent to `C(y)` and the other, where, for some given `n` (an eigenvariable), `B(succ(n),y)` is equivalent to `D(n,y)`. This rule is provable in Isabelle by induction on `x`: the base case and step case follow from the first and second hypothesis respectively.

Now suppose that we have a goal

```
ALL x y. spec(x,y) <-> ?P(x,y)
```

where `spec` is a ground specification (e.g., like *sum*) and `?P` a metavariable. We can use the above rule to reduce the goal to two subgoals

```
ALL y. spec(0,y) <-> ?C(y)
```

and

```
ALL y. spec(succ(n),y) <-> ?D(n,y).
```

Furthermore, this unifies `?P(x,y)` with

```
(x=0 & ?C(y)) | (EX n. succ(n)=x & ?D(n,y)).          (1)
```

This last term meets our syntactic requirement for the body of a Horn clause program, provided `?C(y)` and `?D(n,y)` do too. Hence, we have reduced finding a program `?P` to finding programs `?C` and `?D` that meet simpler specifications. Moreover, when `?C` and `?D` are structurally correct, so is `?P`. When the proof is done, and `?C` and `?D` have been instantiated with ground terms, than so will `?P`. Note too that the same proof rule can be used when `?P(x,y)` is instead a ground term, provided that it unifies with (1).

This proof rule is really a special case of the following induction rule. We will see shortly how this rule is used to create a definition of a Horn program which may (through use of the induction hypothesis given in the last goal) be recursively defined.

```
Induction:
[|  ALL x y z. P(x,y,z) <-> (x=0 & Q(y,z)) | (EX n.succ(n)=x & R(n,y,z));
    ALL y z.S(0,y,z) <-> Q(y,z);
    !!m. ALL y z. S(m,y,z) <-> P(m,y,z) ==> ALL y z.S(succ(m),y,z) <-> R(m,y,z) |]
==> ALL x y z. S(x,y,z) <-> P(x,y,z)
```

It is worth reemphasizing that these rules are formally derived, not asserted as axioms; there is no doubt that they are correct.

We are now ready for an example.

3.1 Example: the sum predicate

Our example is to synthesize the above *sum* predicate. What follows are snapshots taken directly from an Isabelle session (apart from "pretty printing").

The initial goal, as previously indicated, is the equivalence between a ground specification and a metavariable. There is one additional complication: to synthesize recursive programs we need to create a definition that may refer to itself. Isabelle has no facility for creating dynamically new (schematic) definitions during a proof, so instead we allow proof under an assumption `?H` that is to become (incrementally) instantiated with the definition of the (Clark completed) Horn program. Hence, our initial goal is:

```
?H ==> (ALL x y z. x + y = z <-> ?P(x,y,z))
```

We begin by resolving (Isabelle has a resolve tactic which combines higher-order unification with backchaining) the conclusion of this with the induction rule we have previously derived. Examining the previously given induction rule, we see this yields three subgoals; but the first subgoal we unify against `?H`, initializing our schematic definition. Hence the entire proof step (invoked with a tactic that first resolves with the induction rule, then discharges the first assumption through unification with `?H`) reduces the initial goal to the following two subgoals.

```
(ALL x y z. ?P(x,y,z) <-> x = 0 & ?Q(y,z) | (EX n. succ(n) = x & ?R(n,y,z)))
==> (ALL x y z. x + y = z <-> ?P(x,y,z))
1. ALL x y z. ?P(x,y,z) <-> x = 0 & ?Q(y,z) | (EX n. succ(n) = x & ?R(n,y,z))
   ==> ALL y z. 0 + y = z <-> ?Q(y,z)
2. !!m.
   [| ALL x y z. ?P(x,y,z) <-> x = 0 & ?Q(y,z) | (EX n. succ(n) = x & ?R(n,y,z));
      ALL y z. m + y = z <-> ?P(m,y,z) |]
   ==> ALL y z. succ(m) + y = z <-> ?R(m,y,z)
```

Isabelle always prints the goal that is being proved followed by numbered subgoals whose proof is required to establish the goal. Here the top three lines show the goal with `?H` already partially instantiated to a program for `?P`; with each successive snapshot we will see this program further instantiated. The remaining lines are the numbered subgoals and consist of the base and step cases of the induction proof. Note that we are not forced by the system to give a name to the predicate `?P` we are synthesizing.

We now turn our attention to the two cases. Each case will further instantiate the program we are building, the base case instantiating `?Q` (which is the base case of the recursion) and the step case `?R`. The base case is simple: We direct Isabelle's rewriting tactic to apply the appropriate Peano axiom to reduce `0 + y = z` to `y = z`; we prove the resulting simplified subgoal by unifying `?Q(y,z)` with `y = z` which is a Horn body; the equivalence immediately follows. The step case is more complex, and leads to a more involved program construct. Here we resolve against the derived case-split rule. This further contributes to the synthesized program by partially instantiating `?R`.

```
(ALL x y z. ?P(x,y,z) <-> x = 0 & y = z | (EX n. succ(n) = x & (z = 0 & ?C(n,y)
   | (EX na. succ(na) = z & ?D(n,na,y)))))
==> (ALL x y z. x + y = z <-> ?P(x,y,z))
1. !!m.
   [| ALL x y z. ?P(x,y,z) <-> x = 0 & y = z | (EX n. succ(n) = x & (z = 0 & ?C(n,y)
        | (EX na. succ(na) = z & ?D(n,na,y))));
      ALL y z. m + y = z <-> ?P(m,y,z) |]
   ==> ALL y. succ(m) + y = 0 <-> ?C(m,y)
2. !!m n.
   [| ALL x y z. ?P(x,y,z) <-> x = 0 & y = z | (EX n. succ(n) = x & (z = 0 & ?C(n,y)
        | (EX na. succ(na) = z & ?D(n,na,y))));
      ALL y z. m + y = z <-> ?P(m,y,z) |]
   ==> ALL y. succ(m) + y = succ(n) <-> ?D(m,n,y)
```

The case split has yielded two subgoals. In the first, we simplify the conclusion to `ALL y. False <-> ?C(n,y)`. Since `False` is a Horn body, we may unify `?C(m,y)` with it. This expands our program and leaves us with the following singleton subgoal.

```
(ALL x y z. ?P(x,y,z) <-> x = 0 & y = z | (EX n. succ(n) = x &  (z = 0 & False
   | (EX na. succ(na) = z & ?D(n,na,y)))))
==> (ALL x y z. x + y = z <-> ?P(x,y,z))
1. !!m n.
   [| ALL x y z. ?P(x,y,z) <-> x = 0 & y = z | (EX n. succ(n) = x & (z = 0 & False
        | (EX na. succ(na) = z & ?D(n,na,y))));
      ALL y z. m + y = z <-> ?P(m,y,z) |]
   ==> ALL y. succ(m) + y = succ(n) <-> ?D(m,n,y)
```

Rewriting simplifies `succ(m) + y = succ(n)` to `m + y = n` and the goal becomes:

```
(ALL x y z. ?P(x,y,z) <-> x = 0 & y = z | (EX n. succ(n) = x &  (z = 0 & False
   | (EX na. succ(na) = z & ?D(n,na,y)))))
==> (ALL x y z. x + y = z <-> ?P(x,y,z))
1. !!m n y.
   [| ALL x y z. ?P(x,y,z) <-> x = 0 & y = z | (EX n. succ(n) = x & (z = 0 & False
        | (EX na. succ(na) = z & ?D(n,na,y))));
      ALL y z. m + y = z <-> ?P(m,y,z) |]
   ==> m + y = n <-> ?D(m,n,y)
```

And now the conclusion may be resolved against the induction hypothesis

```
ALL y z. m + y = z <-> ?P(m,y,z).
```

Using the induction hypothesis this way gives us a recursive program as it invokes a call to the predicate we are defining. This completes the proof and Isabelle yields the following proven sequent with no subgoals.

```
(ALL x y z. ?P(x,y,z) <-> x = 0 & y = z | (EX n. succ(n) = x & (z = 0 & False
   | (Ex na. succ(na) = z & ?P(n,y,na)))))
==> (ALL x y z. x + y = z <-> ?P(x,y,z))
```

Taking stock

Let us review what has been accomplished and what the example demonstrates. First through our proof we have constructed the Horn program `?P(x,y,z)` given in the final proof state and proved it correct. Program instantiation occurred only through resolution and the same proof would have worked if we had given the program `?P` up front. So there is no difference between verification and synthesis here. This is true for any proof similarly developed.

Second, the resulting proof is guaranteed to be logically correct. This follows as every rule used to build the proof is either a primitive rule in our theory or a derived rule. This point deserves further clarification though. To handle the problem of defining a recursive predicate, we set up the proof within an assumed "context": the hypothesis `?H`, which is instantiated with the recursive definition or definitions. Naturally, if the context is inconsistently instantiated the proof, while still correct, will not give rise to a verified program. Our emphasis in this paper is on the schematic proof mechanism itself and its flexibility — there are means, of course, to ensure consistency of contexts depending on the logic in question. In our logic we could check consistency afterwards analogous to what occurs after synthesis in INKA (see section 4) or the way recursive definitions are checked in NQTHM. Another way would be to include in the logic a means of establishing through proof that the definition is sensible (e.g., well-founded). Such an interactive obligation would be easier in type theory, for example, where recursive definitions can be given using fix-point combinators and termination shown via appropriate inductions.

A final point is that the object constructed satisfies our structural requirement: it is a Horn-clause program.[3] As with context consistency, this is a meta-level problem in the sense that we do not enforce it in our logical theory. (Though there are theories, of course, that do have well-formedness proof obligations, e.g., Martin-Löf's type theories.) In our case, structural correctness is particularly easy to understand and (at the meta-level) to ensure. That is, if we restrict proof steps that instantiate the program so that they use our derived rules, the induction hypothesis, or atomic rules, then the objects synthesized will always meet our structural requirement. This invariant follows by induction on the structure of derivations using these rules.

4 Related Work

In this section we will look at a selection of other work in program synthesis/transformation and relate that work to the approach we propose.

[3] Not quite as unification never forced us to *name* the synthesized predicate `?P`. But the result is a Horn program under any predicate name.

Resolution Based Synthesis

In [13] Green suggests using the "answer predicate trick" in resolution (see also [9] for a description of this trick) to find unifiers using resolution to answer questions. In this setting, as in Prolog, object-level variables stand in for meta-variables and are assigned values, possibly incrementally, using resolution.

Let $R(x, y)$ be a first-order relation between x and y. Green's contribution is to show how resolution can serve as a basis for answering each of the following types of questions:

Problem	Question	Desired Answer
checking	$R(a, b)$	yes or no
simulation	$\exists x.\, R(a, x)$	yes $x = b$ or no
verifying	$\forall x.\, R(x, g(x))$	yes or no $x = c$
programming	$\forall x.\, \exists y.\, R(x, y)$	yes $y = f(x)$ or no $x = c$

The first three are straightforward, but the last may be surprising; it is directly related to our proposal. To see if $\forall x.\, \exists y.\, R(x, y)$ follows from a set of formulae, Green negates it, turns it into a clause, and searches for a refutation. Negated and clausified it becomes $\{\neg R(x_0, y)\}$, and if we can prove inconsistency we get a unifier for y, e.g., $y = car(x_0)$. Hence we have y as a "function" of x_0, and only first-order variables and unification have been used in the proof process. Green can get away with using first-order variables because skolemizing removes binding operators and, at the same time extends (implicitly) the signature (e.g., with x_0) in which the theorem is proved. He is really not synthesizing a function, but a first-order object. However, since the context variables are arbitrary, he can generalize afterwards.

We see here the essence of the schematic proof idea in a simple first order setting: the same resolution steps can be used either to verify that a function is correct, or to construct a function.

Deductive Synthesis

There are a number of approaches loosely classified under the heading of deductive synthesis, which also have close ties to schematic proof. In Manna and Waldinger's deductive tableau system, [25], one proves a goal $\forall\, \overline{x}.\exists\, \overline{y}.R(\overline{x}, \overline{y})$ by manipulating $R(\overline{x}, \overline{y})$ in a "tableau" where the $\overline{x}$ are turned into eigenvariables and the $\overline{y}$ are kept in "output columns", essentially as metavariables. This is quite similar to Green's approach, except that the goal is not negated, since the proof is not refutation based. Similarly to Green's and our work, each proof step may extend the output metavariables using substitutions derived by rule application. At the end of the proof, if variables have been instantiated only by primitive program constructors (this is analogous to our side conditions), the tableau yields a completed program. Again, this is very similar to the schematic proof idea, and, as with Green's work, they are able to operate on quantifier free terms, so they can use first-order metavariables in their proofs (later implicitly generalizing the first-order object they have constructed to a program).

Another approach to deductive synthesis, proposed for example by Biundo [2,3], and Steinbrüggen [30] is, given a formula $\phi = \forall \overline{x}.\, \exists y.\, p(\overline{x}, y)$, to prove instead the skolemized version $\phi_0 = \forall \overline{x}.\, p(\overline{x}, f(\overline{x}))$. Proving ϕ_0 is not only sufficient to prove ϕ, but it provides us with a recursive function f that meets the specification p as well. In Biundo's work, proof proceeds top down, with f an uninterpreted function constant. At some point the system has isolated, in various "subgoals", $f(x, y)$ as conditional equations. For example, $\forall x\, y.\, y = 0 \rightarrow f(x, y) = x$ and $\forall x\, y\, v.\, y = succ(v) \rightarrow f(x, y) = succ(f(x, v))$ arise in the synthesis of the plus function (given a definition for subtraction). These equations can be accepted as a definition for f after side conditions are checked (e.g., that f terminates, is deterministic, and is defined on all inputs). Note that this approach, unlike those previously discussed, does not really incrementally "instantiate" f. Instead, the definition of f is given to the system (and side conditions are checked) all at once. Incremental development here is problematic since we cannot view these equations as saying how to assign a unique term in the object logic to f; indeed, no such term exists! However, if instead of using her equational representation of functions, they existed as terms in her object language, like in Boyer and Moore's logic [4], then another option would be available to her. Under such a representation it would be possible to view an equation involving a skolem function as specifying an equality between a schematic function and its specification. This would be analogous to the relationship between schematic Horn programs and their specifications in our logic programming example (where $\leftrightarrow$ plays the role of equality). In such a setting, her skolem functions could be replaced by metavariables and instantiated incrementally.

A final example of a deductive synthesis approach, and one which clearly exemplifies the schematic proof idea, is that of the KIV theorem prover. KIV is a prover for dynamic logic that supports proofs with metavariables. Within dynamic logic one can formulate other logics such as Hoare logic, when we could then manipulate formulae like $\{P\}\, S\, \{Q\}$. As with our approach, any of these components can be partially or completely uninstantiated (by using metavariables) during the proof. KIV uses first-order metavariables (see comments in section 2) but we believe the work would be better served by implementation in a system like Isabelle which supports higher-order schematic proofs, while providing the same kind of tactic based metareasoning support as currently.

Constructive Type Theories

As explained in the introduction, our schematic proof idea can be seen to capture the kind of program construction that takes place in many type theories. The common explanation of program extraction in a type theory like Nuprl's is that, as the logic is constructive, a proof yields a program which exhibits the implicit construction. This is correct but the emphasis can be misleading. As indicated previously, the schematic proof idea is enough to account for extraction in type theories like Martin-Löf's or Nuprl's; so constructivity is not significant in *extraction*, but rather *execution*. That is, because the logic is constructive, the terms extracted can be effectively executed and their evaluation behavior

agrees with the semantics of the type theory. We could extend such languages to extract constructions from classical proofs (e.g., see [18]), but we would not be able to execute the results since, for example, there is no general way to decide which branch of a case-split to execute when this corresponds to a use of the law of the excluded middle.

Hardware Synthesis

We conclude with an example from the domain of hardware synthesis; what is interesting about this example is that it presents an idea based on "validation functions" and "achievement theorems" which seems rather different from the schematic proof idea but can be cleanly reformulated in our framework. The reformulation yields a conceptually and implementationally simpler calculus than the original presentation.

The Veritas group at Kent [14,15] proposed a novel approach to synthesizing circuits. Their technique, which they call *Formal Synthesis*, is to refine a "design goal" (*spec*) interactively, using *methods*, eventually concluding with a theorem that some circuit specification *circ* achieves *spec*; i.e., that $circ \rightarrow spec$. Each method consists of a pair of functions: a subgoal function and a validation function. The former decomposes a specification *spec* into subspecifications $spec_i$. The latter constructs from a proof that the $circ_i$ achieve $spec_i$, an implementation *circ* with a proof that *circ* that achieves *spec*.

The relationship between implementation and specification is different from the one we used for logic program synthesis (implication instead of equivalence) but the idea is the same: apply rules to decompose the specification and each rule should tell how to construct an implementation that stands in the desired relation to the specification. An example should make this clear, as well as our recasting.

The *Split* method takes a specification which is the conjunction of specifications and returns the individual specifications as subgoals.

$$\frac{spec_1 \quad spec_2}{spec_1 \wedge spec_2} \qquad (Split)$$

The validation theorem for this is the following inference rule[4]

$$\frac{circ_1 \rightarrow spec_1 \quad circ_2 \rightarrow spec_2}{(circ_1 \wedge circ_2) \rightarrow (spec_1 \wedge spec_2)}$$

Looking at the above validation theorem we immediately see that this is a derivable rule in Isabelle (in a first or higher-order theory). Their other rules

[4] In the hardware formal reasoning community, circuits are often represented as relations expressing constraints (on port values) [8]. These constraints may be joined by logical $\wedge$ and this also represents an operator that builds a circuit from two subcircuits by parallel composition. So the $\wedge$ used to construct the implementation in the validation theorem represents both a circuit constructor (so we see we are building a circuit) and a logical operator (so we may prove facts about it).

are more complex, but they too are all derivable — the authors point this out themselves. Therefore the recasting of their idea as a schematic proof approach is especially simple. We may do away with their machinery of subgoaling and validation functions; instead we start simply with a schematic achievement theorem where the circuit is given by a metavariable, and directly apply their derived rules as schematic inference rules. As with the rules for logic program synthesis, the entire development calculus can be simply and formally developed within Isabelle, giving a formal guarantee of correctness not just to the proofs but to the calculus itself. We are currently experimenting with an implementation based on these ideas.

5 Conclusion

We have given a general framework for unifying the ideas of synthesis and verification; we have shown how it can be applied to formalize and use new calculi as well as understand and simplify a range of previously proposed approaches.

Acknowledgements

The first author thanks Tobias Nipkow for extensive help and encouragement using Isabelle.

References

1. David A. Basin. Extracting circuits from constructive proofs. In *IFIP-IEEE International Workshop on Formal Methods in VLSI Design*, Miami, USA, January 1991.
2. S. Biundo. Automated synthesis of recursive algorithms as a theorem proving tool. In *Eighth European Conference on Artificial Intelligence*, pages 553–8. Pitman, 1988.
3. S. Biundo. Plan generation using a method of deductive program synthesis. Research Report RR-90-09, DFKI, 1990.
4. Robert S. Boyer and J. Strother Moore. *A Computational Logic*. Academic Press, 1979.
5. A. Bundy. Tutorial notes; reasoning about logic programs. In *Logic programming in action*, pages 252–277. Springer Verlag, 1992.
6. A. Bundy, F. van Harmelen, C. Horn, and A. Smaill. The Oyster-Clam system. In *10th International Conference on Automated Deduction*, pages 647–648. Springer-Verlag, 1990.
7. R.M. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of the Association for Computing Machinery*, 24(1):44–67, 1977.
8. Albert Camilleri, Mike Gordon, and Tom Melham. Hardware verification using higher-order logic. In *From HDL Descriptions to Guaranteed Correct Circuit Designs*, pages 43–67. Elsevier Science Publishers B. V. (North-Holland), 1987.
9. C-L. Chang and R. C-T. Lee. *Symbolic logic and mechanical theorem proving*. Academic Press, 1973.

10. Robert L. Constable et al. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice Hall, 1986.
11. Thierry Coquand and Gérard Huet. The Calculus of Constructions. *Information and Computation*, pages 95–120, 1988.
12. Amy Felty and Dale Miller. Specifying theorem provers in a higher-order logic programming language. In *9th International Conference On Automated Deduction*, Argonne, Illinois, 1988.
13. Cordell Green. Application of theorem proving to problem solving. In *Proceedings of the IJCAI-69*, pages 219–239, 1969.
14. F. K. Hanna, M. Longley, and N. Daeche. Formal synthesis of digital systems. In *IMEC-IFIP International Workshop on: Applied Formal Methods For Correct VLSI Design*, volume 2, pages 532–548, Leuven, Belgium, 1989.
15. F.K. Hanna, N. Daeche, and M. Longley. VERITAS+: A specification language based on type theory. In *Hardware Specification, Verification and Synthesis: Mathematical Aspects*, Ithaca, New York, 1989. Springer-Verlag.
16. M. Heisel, W. Reif, and W. Stephan. Tactical theorem proving in program verification. In *10th International Conference On Automated Deduction*, pages 117–131, Kaiserslautern, FRG, 1990.
17. Jane Hesketh, Alan Bundy, and Alan Smaill. Using middle-out reasoning to transform naive programs into tail recursive ones. In *Proceedings of CADE-11*, 1992.
18. Douglas J. Howe. On computational open-endedness in Martin-Löf's type theory. In *Sixth Annual Symposium on Logic in Computer Science*, Amsterdam, 1991.
19. Ina Kraan, David Basin, and Alan Bundy. Logic program synthesis via proof planning. In *Logic Program Synthesis and Transformation*, pages 1–14. Springer-Verlag, 1993.
20. Ina Kraan, David Basin, and Alan Bundy. Middle-out reasoning for logic program synthesis. In *10th International Conference on Logic Programing (ICLP93)*, 1993.
21. J.W. Lloyd. *Foundations of Logic Programs*. Symbolic Computation. Springer-Verlag, 1987. Second, extended edition.
22. J.W. Lloyd and R.W. Topor. Making Prolog more expressive. *J. Logic Programming*, 1(3):225–240, 1984.
23. Z. Luo and R. Pollack. Lego proof development system: User's manual. Report ECS-LFCS-92-211, Department of Computer Science, University of Edinburgh, May 1992.
24. M.J. Maher. Equivalences of logic programs. In *Foundations of Deductive Databases and Logic Programming*. Morgan Kaufmann, 1987.
25. Z. Manna and R. Waldinger. A deductive approach to program synthesis. *ACM Transactions on Programming Languages and Systems*, 2(1):90–121, 1980.
26. Per Martin-Löf. Constructive mathematics and computer programming. In *Sixth International Congress for Logic, Methodology, and Philosophy of Science*, pages 153–175, Amsterdam, 1982. North Holland.
27. Dale Miller. Abstractions in logic programs. In *Logic and Computer Science*, pages 329–359. Academic Press, 1990.
28. L.C. Paulson. Isabelle: the next 700 theorem provers. In *Logic and Computer Science*, pages 77–90. Academic Press, 1990.
29. Frank Pfenning. Logic programming in the LF logical framework. In *Logical Frameworks*, pages 149 – 181. Cambridge University Press, 1991.
30. Ralf Steinbrüggen. Programs viewed as SKOLEM functions. In *Methods of Programming, Selected Papers on the CIP-Project*, LNCS, pages 125 – 134. Springer-Verlag, 1991.

31. Carolyn Talcott. A theory of binding structures, and applications to rewriting. *TCS* volume 112, 1993.